Unit –2: Function, Structure and Working with Object

Function

- ➢ A function is a block of code that performs a specific task.
- ➢ Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

There are two types of function:

1. Standard Library Functions: Predefined in C++ header files such as ceil(x), cos(x), exp(x), etc
2. User-defined Function: Created by users

Declaration of a function

- The **syntax** to declare a function is:

```
returnType functionName (parameter1, parameter2,...) {
    // function body
}
```

Example

```
void greet()
{
    cout << "Hello World";
}
```

Calling a Function

- In the above program, we have declared a function named greet(). To use the greet() function, we need to call it.
- Here's how we can call the above greet() function.

```
int main()
{
    greet();
}
```

Function Parameters

- Function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.
- For example, let us consider the function below:

```
void printNum(int num)
{
      cout << num;
}
```

```
#include <iostream>
using namespace std;
void displayNum(int n1, float n2)
 {
          cout << "The int number is " << n1;
          cout << "The double number is " << n2;
}
int main()
{
          int num1 = 5;
         double num2 = 5.5;
         displayNum(num1, num2);
        return 0;
}
```

Function Prototype

- In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype. For example,
- This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.
- The syntax of a function prototype is:

returnType functionName(dataType1, dataType2, ...);

```
#include <iostream>
using namespace std;
int add(int, int);
int main()
 {
        int sum;
        sum = add(100, 78);
        cout << sum << endl;
        return 0;
}
int add(int a, int b) {
   return (a + b);
}
```

Parameter

Formal Parameter

- Parameter Written in Function Definition is Called “Formal Parameter.
- Formal parameters are always variables, while actual parameters do not have to be variables.

Actual Parameter

- Parameter Written in Function Call is Called “Actual Parameter”.
- One can use numbers, expressions, or even function calls as actual parameters.

Example

```
void display(int para1)
{
        printf( “ Number %d “ , para1);
}
void main()
{
        int num1;
        display(num1);
}
```

In above, **para1 is called the Formal Parameter**

num1 is called the Actual Parameter

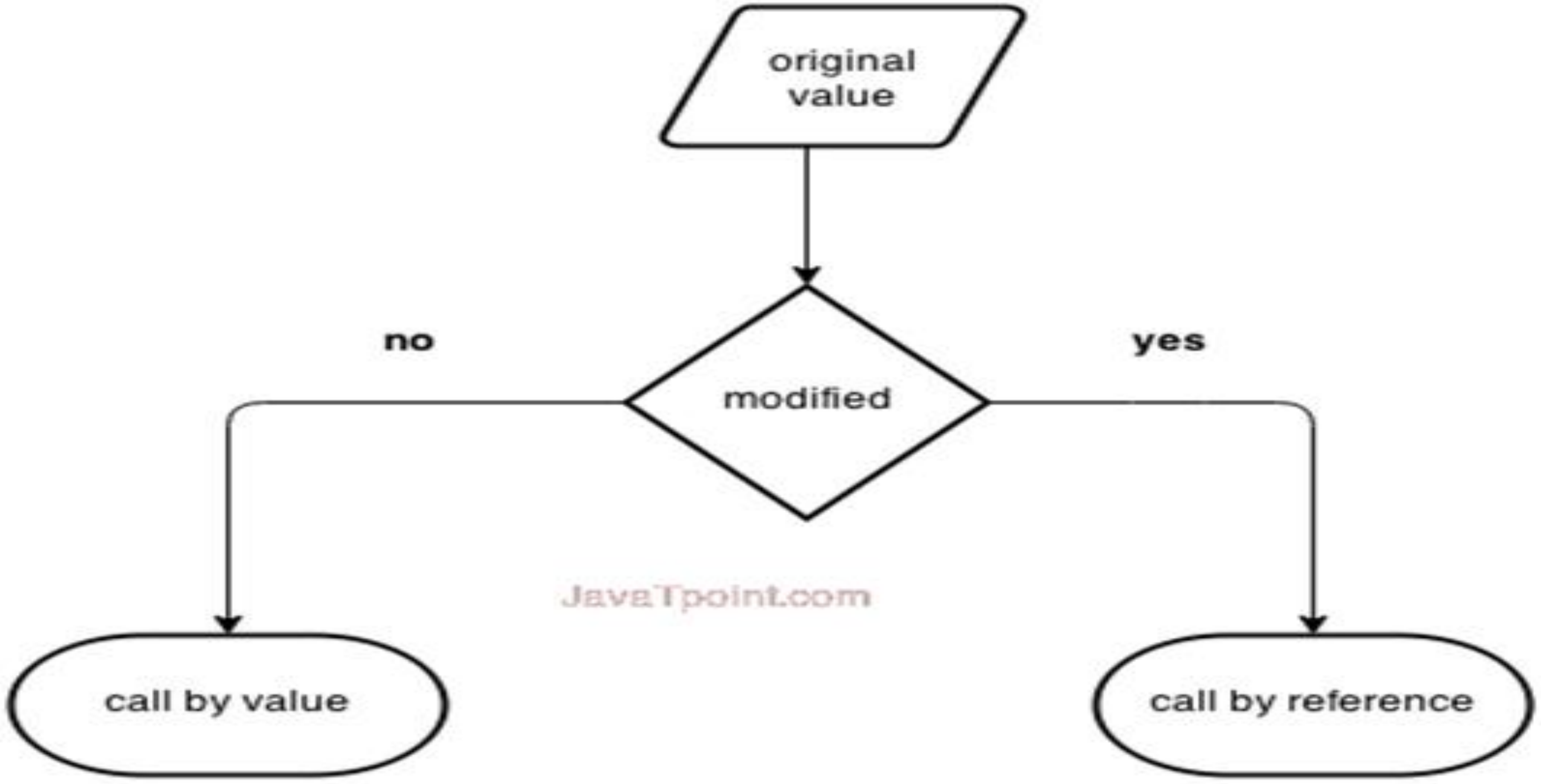
Categories of function

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

1. Function with no argument and no return value
2. Function with no argument but return value
3. Function with argument but no return value
4. Function with argument and return value

Call by value and call by reference

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Call by Value

```
#include <iostream>
using namespace std
void swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
}
int main() {
 int a = 40, b = 50;
 int cout << "Before swap: a = " << a << " b = " << b << endl;
 swap(a, b);
 cout << "After swap: a = " << a << " b = " << b << endl;
 return 0;
}
```

Output

```
Before swap: a = 40 b = 50
After swap: a = 40 b = 50
```

Call by Reference in

```
#include <iostream>
using namespace std;
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
         *b = temp;
}
int main() {
 int x = 5;
 int y = 10;
 cout << "Before swap: x = " << x << " , y = " << y << endl;
 swap(&x, &y);
 cout << "After swap: x = " << x << ", y = " << y << endl;
 return 0;
}
```

Output

```
Before swap: x = 5, y = 10
After swap: x = 10, y = 5
```

Function Overloading in C++

- Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters.
- When a function name is overloaded with different jobs it is called Function Overloading.
- In Function Overloading “Function” name should be the same and the arguments should be different.
- Function overloading can be considered as an example of a polymorphism(compile time) feature in C++.

- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.
- The parameters should follow any one or more than one of the following conditions for Function overloading:
- Parameters should have a different type

  add(int a, int b)

  add(int a, int b)
- functions may or may not have different return types but they must have different arguments. For example,

  // Error code

  int test(int a) { }

  double test(int b){ }

```
#include <iostream>
void display(int var1, double var2)
{
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}
void display(double var) {
    cout << "Double number: " << var << endl;
}
void display(int var){
    cout << "Integer number: " << var << endl;
}
int main() {
    int a = 5;
    double b = 5.5;
        display(a);
        display(b);
        display(a, b);
    return 0;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and double number: 5.5
```

Rules of Function Overloading in C++

1. The functions must have the same name
2. The functions must have different types of parameters.
3. The functions must have a different set of parameters.
4. The functions must have a different sequence of parameters.

Recursion

- Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.
- Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Example

```
int sum(int k)
{
        if (k > 0) {
         return k + sum(k - 1);
        } else {
        return 0;
  }
}
int main() {
  int result = sum(10);
  cout << result;
  return 0;
}
```

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Inline function

- One of the key features of C++ is the inline function
- If make a function is inline, then the compiler replaces the function calling location with the definition of the inline function at compile time.
- Any changes made to an inline function will require the inline function to be recompiled again because the compiler would need to replace all the code with a new code; otherwise, it will execute the old functionality.
- When an inline function is invoked, its entire body of code is added or replaced at the inline function call location. At compile time, the C++ compiler makes this substitution.

Inline function

The compiler may not implement inlining in situations like these:

1. If a function contains a loop. (for, while, do-while)
2. if a function has static variables.
3. Whether a function recurses.
4. If the return statement is absent from the function body and the return type of the function is not void.
5. Whether a function uses a goto or switch statement.

difference between the normal function and the inline function.

- Inside the main() method, when the function fun1() is called, the control is transferred to the definition of the called function. The addresses from where the function is called and the definition of the function are different. This control transfer takes a lot of time and increases the overhead.
- When the inline function is encountered, then the definition of the function is copied to it. In this case, there is no control transfer which saves a lot of time and also decreases the overhead.

```
#include <iostream>
using namespace std;
inline int add(int a, int b)
{
    return(a+b);
}
int main()
{
    cout<<"Addition of 'a' and 'b' is:"<<add(2,3);
   return 0;
}

// cout<<"Addition of 'a' and 'b' is:"<<return(2+3);
```

Default arguments

- A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

Following are the rules of declaring default arguments –

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Default arguments

Example

void function(int x, int y, int z = 0)

Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.

Void function(int x, int z = 0, int y)

Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

Example

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}
int main()
{
    cout << sum(10, 15) << endl;             // x = 10, y = 15, z = 0, w = 0
    cout << sum(10, 15, 25) << endl;         // x = 10, y = 15, z = 25, w = 0
    cout << sum(10, 15, 25, 30) << endl;     // x = 10, y = 15, z = 25, w = 30
    return 0;
}
```

C++ Structure

- C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.
- The struct keyword defines a structure type followed by an identifier (name of the structure).
- When a structure is created, no memory is allocated.

Example:

```
struct person
{
        char name[50];
        int age;
        float salary;
}p1,p2;
```

Defining structure variable

- Structure variables can be declared by two way

1) with structure declaration

Example:

```
    struct student
    {
            int rollno;
            char name[50];
            int marks;
    }s1,s2;
```

Here, s1 and s2 are structure variable.

2) as a separate declaration inside or outside main( )

Example:

```
struct student
{
        int rollno;
        char name [50];
        int marks;
};
void main( )
{
        struct student s1,s2;
}
```

Accessing structure members

- To access any member of a structure, we use the member access operator (.).
- The member access operator is coded as a dot between the structure variable name and the structure member that we want to access.

Syntax:

```
structure_variable.member_name
```

Example:

```
struct student
{
        int rollno;
        char name [50];
        int marks;
};
void main( )
{
        struct student s1,s2;
        s1.rollno=1;            // accessing member rollno
        s1.name=”ABC”;      // accessing member name
        s1.marks=60;          // accessing member marks
}
```

Introduction to class and object

Class

- Class is a collection of objects.
- In class we can bind the data (variable) and its associated functions together.
- It is called data encapsulation.
- Class allows the data and function to be hidden if necessary.
- When we create a class it treat as a built in data type.
- So, we can create the variable of that data-type, which is called as objects.
- A class specification has generally two parts:

1.Class declaration 2. Class function definitions

- The class declaration describes the type and scope of its members.
- The class function definition describes how the class functions are implemented.
- To declare the class we have to use the class keyword.

Objects:

- Object is a basic run-time entity in object oriented system.
- Object is a one type of class variable.
- Object can be declared as same as the variable declaration.

Declaration of class

To declare the class we have to use the class keyword.

Syntax of class declaration:

```
class class-name
{
        private:
                variable declarations;
                function declarations;
        public:
                variable declarations;
                function declarations;
};
```

- The variable which is declared inside the class declaration that is known as **data members.**
- The functions which is declared inside the class declarations that is known as **member functions.**
- The data member and member functions are also known as the **class members**.
- **Private** and **public** specifies the scope (where it is used) of the class members.
- By default, the scope of the class members is **private**.
- The class members which are declared as private can be accessed only from within the same class.
- The class members which are declared as public can be accessed from outside the class also.

<u>Example of class declaration:</u>

```
class student
{
        int no;
        char name[20];
        public:
                void getdata( );
                void putdata( );
};
```

Here, no and name are the data members which has the private scope.

And getdata( ), putdata( ) are the member functions which has the public scope.

Declaration of object:

Objects can be declared by two way

1) with class declaration

Objects can be created when a class is defined by placing their names immediately after the closing brace, same as structures.

Example:

```
class student
{
        . . . . .
        . . . . .
}s1,s2,s3;
```

Where s1,s2 and s3 are the object name.

2) as a separate declaration inside or outside main( ).

- Objects can be created after declaration of class.
- It can be declared inside main( ) function.

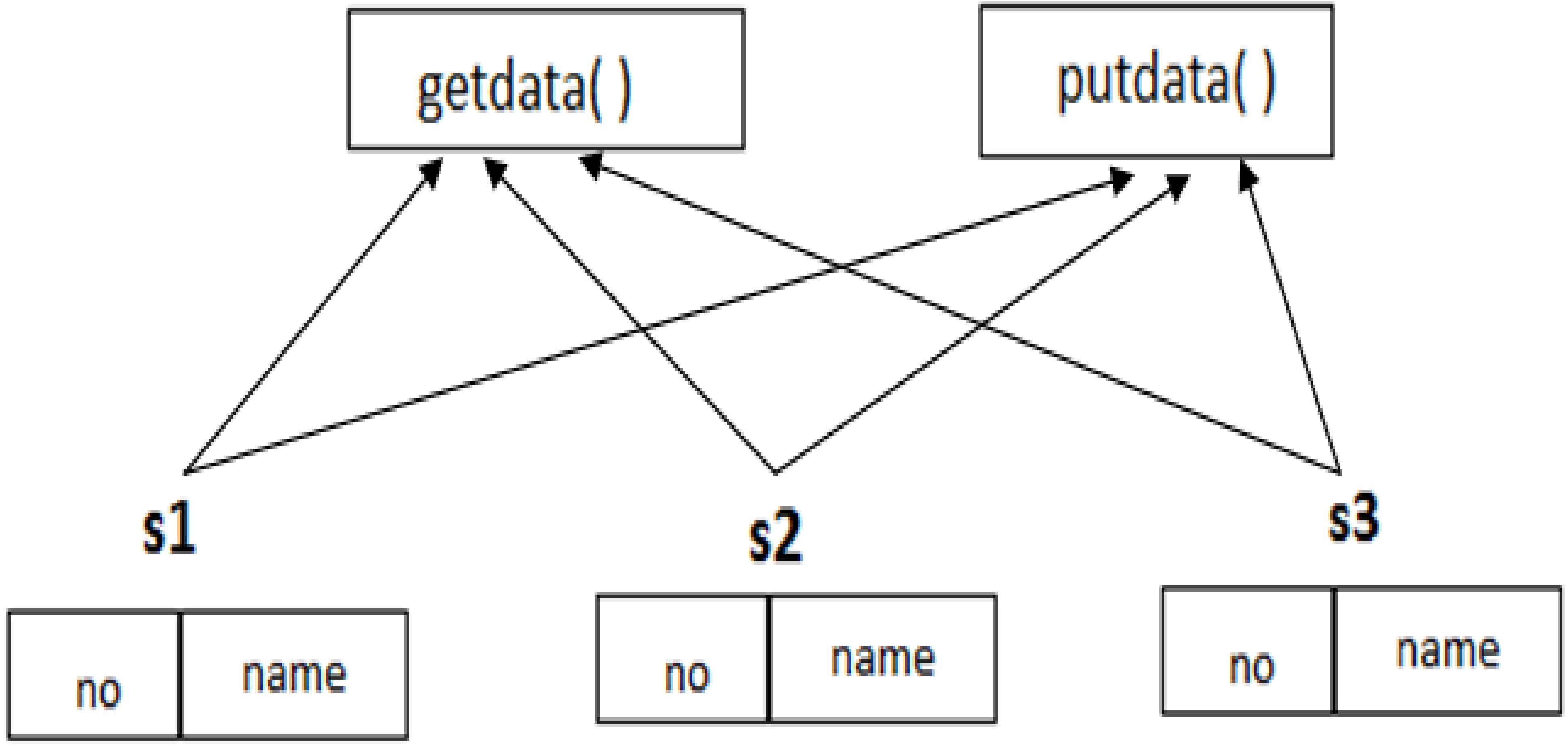
Syntax:

```
class-name object-name;
```

Example:

```
student s1,s2,s3;
```

where, student is the class name and s1,s2 and s3 are the object name.


getdata( )
putdata( )
s1
s2
s3
no
name
no
name
no
name

Access Specifier-Private, public and protected

- Access specifiers define how the members (variable or function) of a class can be accessed.
- In C++, there are three access specifiers:

  public - members are accessible from outside the class

  private - members cannot be accessed (or viewed) from outside the class

  protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.
- By default, all members of a class are private if you don't specify an access specifier:

<u>Example:</u>

```
class student
{
	int no;
	char name[20];
	private:
	void getdata( )
	{
		cout<< “Enter Numnber and Name :”;
		cin>>no>>name;
	}
	void putdata( )
	{
		cout<< “Number : “<<no<<” \nName
:”<<name;
	}
};
```

```
void main( )
{
	Student s1;
	s1.getdata( );
	s1.putdata( );
	Cout<<no;
}
```

Defining member function inside

1. **Inside the class definition:**

Example:

```
class student
{
        int no;
        char name[20];
        public:
                void getdata( );
                void putdata( )
                 {
                        cout<< “Number : “<<no<<” Name :”<<name;
                }
};
```

2. Defining member function outside of the class using scope resolution operator

- Member function can be declared outside the class, if it is only declared inside class.
- Definition of member function are like normal function
- Whenever the definition of class member function outside the class, the member name must be qualified by the class name using the scope resolution operator.

Syntax:

```
return_type class_name::function_name(Arguments)
{
        Statements;
}
```

Example:

```
class student
{
        int no;
        char name[20];
        public:
        void getdata( );
        void putdata( ) // definition inside class
        {
                cout<< “Number : “<<no<<” Name :”<<name;
        }
};
void student::getdata( )
{
        cout<<”Enter Number: “;
        cin>>no;
        cout<<”Enter Name: “;
        cin>>name;
}
```

Private member function

- A function declared inside the private access specifier of the class, is known as a private member function.
- Normally we define data members as private and function member as public.
- But in some cases we require to declare function as private.
- That private member function is allowed to access within class only.

Example:

```
class array
{
        private:
        int a[5];
        void swap(int i,int j)
        {
                int temp;
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
        }
        Public:
        void getarray( )
        {
                for(int i=0,i<5;i++)
                cin>>a[i];
        }
void sort( )
{
        for(int i=0,i<4;i++)
        {
                for(int j=i+1;j<5;j++)
                {
                        if(a[i]>a[j])
                        swap(i,j);
                }
        }
}
void putarray( )
{
        for(int i=0,i<5;i++)
        cout<<a[i];
}
};
void main( )
{
        array a1;
        a1.getarray( );
        a1.sort( );
        a1.putarray( );
}
```

Outside member function as inline

- C++ provides inline functions to reduce the function call overhead.
- We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of the function definition.

Example

```
class item
{
            ......
            public: void getdata(int a,float b);
};
inline void item :: getdata(int a,float b)
{
            number=a;
            cost=b;
}
```

Static member and member function

Static Data member

- A data member, marked static, is not associated with any object of that class. Instead, that member is associated with the class itself.
- When we define the data member of a class using the static keyword, the data members are called the static data member.
- A static data member is similar to the static member function because the static data can only be accessed using the static data member or static member function.
- **It is initialized to zero when the first object of its class is created. No other initialization is permitted.**
- **Only one copy of that member is created for the entire class and it is shared by all the objects of that class.**

- It is visible only within the class, but its lifetime is the entire program.
- Static variables are normally used to maintain values common to the entire class.
- **The definition of the static data member is written outside the class.**

Syntax of definition of static data member:

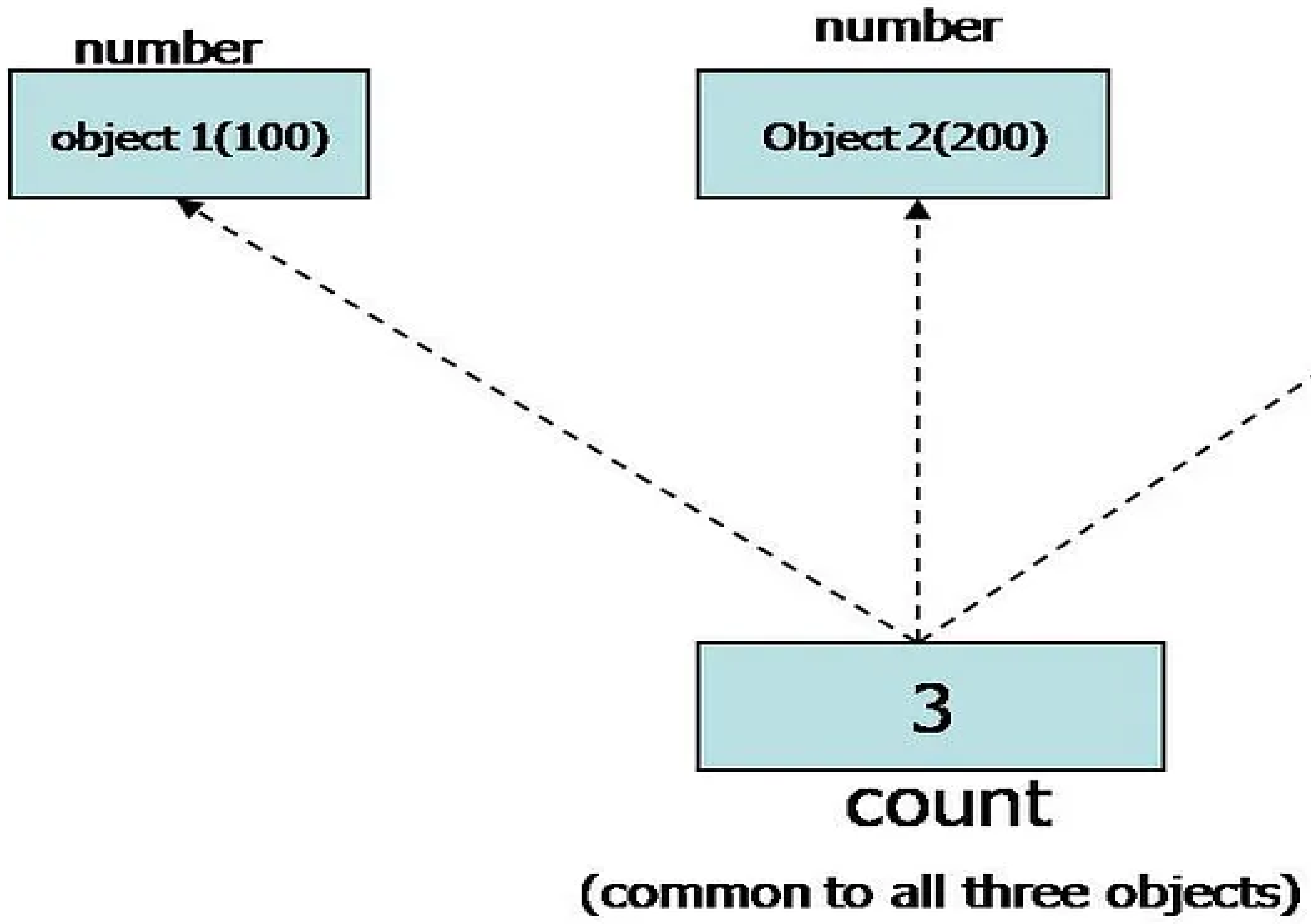
data-type class-name:: variable-name= value;

where the value is optional.

Example:

int item :: count;

where, item is the class-name and count is the static data member.


number
object 1(100)
number
Object 2(200)
number
Object 3(300)
3
count
(common to all three objects)

```
class MyClass
{
        public:
        static int x;
        int y;
         void print()
        {
                x = x + 10;
                y=y+10;
                cout << x << endl;
                cout << y << endl;
        }
};
int MyClass::x = 0;
```

```
int main()

{

  MyClass obj1, obj2, obj3;

  obj1.print();

  obj2.print();

   obj3.print();

cout<<Myclass::x;

}
```

Output:

```
x = 10  y=10
x = 20  y=10
x = 30  y=10
```

Static Member Functions

- The member function of the class which is declared with the static keyword, it is called as static member functions.
- Static member has the following characteristics:
- **A static member function can have access to only other static members declared in the same class. Static members can be either data member or member function.**
- **A static member function can be called using the class name as follows:**
- **class-name :: function-name;**

```
class test
{
        int code;
        static int count;
        public:
        void setcode(int );
        void showcode( );
        static void showcount( )
        {
                cout<<count<<endl;
        }
};
int test::count=0;
void test ::setcode(int a)
{
        code=a;
 }
```

```
void test ::showcode()
{
        cout<<code<<endl;
}
int main()
{
        test t1;
        t1.setcode(10);
        t1.showcode();
        t1.showcount();
        //test::showcount();
}
```

Output:

```
10
0
```

Array of object

- An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object.
- An array is a collection of similar type, therefore an array can be a collection of class type.
- An array of objects is declared in the same way as an array of any built-in data type

Syntax: class_name array_name [size] ;

```
class books
{
          char tit1e[30];
          float price ;
          public:
          void getdata ()
          {
                    cout<<"Title:”;
                    cin>>title;
                    cout<<"Price:”;
                    cin>>price;
          }
          void putdata ()
          {
          cout<<"Title:"<<title<< "\n";
          cout<<"Price:"<<price<< "\n”;
          }
} ;
```

```
void main ()
{
books b[3] ;
for(int i=0;i<3;i++)
{
          cout<<"Enter details o£ book "<<(i+1)
          b[i].getdata();
}
for(int i=0;i<3;i++)
{
          cout<<"\nBook "<<(i+l)<<"\n";
          b[i].putdata() ;
}
getch( );
}
```

Object as a function argument

- The objects of a class can be passed as arguments to member functions as well as non-member functions either by value or by reference.
- When an object is passed by value, a copy of the actual object is created inside the function.
- Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator.

```
class Example
{

        public:
        int temp;
        void readData(int i)
        {
                temp=i;
        }
        Example copy( Example obj)
        {
                return obj;
        }
};
```

```
void main( )
{
        Example e1,e2,e3;
        e1.readData(10);
        e2.readData(20);
        e3=e1.copy(e2);
        cout<<e1.temp<<endl;
        cout<<e2.temp<<endl;
        cout<<e3.temp<<endl;
}
```

Output:

```
10
20
20
```

```
class student
{
        public:
        int age;
        void getAverage(int a)
        {
                age = a;
        }
        void calAverage(student s1, student s2)
        {
        int average = (s1.age + s2.age) / 2;
        cout << average;
        }
};
```

```
int main()
{
        student s1, s2;
        s1.getAverage(10);
        s2.getAverage(20);
        s1.calAverage(s1, s2);
        return 0;
}
```

Output:
```
15
```

Friend Function

- To make an outside function friendly to a class. We have to declare this function as a friend of the class.
- The function declaration should be preceded by the keyword friend.
- The functions that are declared with the keyword friend are known as friend function.
- The function is defined elsewhere in the program like a normal C++ function.
- Usually it has the object as arguments.

Characteristics of the Friend Function:

- It is not in the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of the object.
- It cannot access the member names directly and has to use an object name and membership operator with each member name.
- It can be declared either in the public or private part of a class.
- **Once a function is declared as friend in the class that function must be defined outside the class.**
- **The function definition does not use either the keyword friend or scope resolution operator.**

```
class Box
{
        private:
        int length;
        public:
        void set(int a)
        {
                length=a;
        }
        friend int printLength(Box);
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
```

```
int main()
{
        Box b;
        b.set(10);
         cout<<"Length of box: “;
        cout<< printLength(b)<<endl;
        return 0;
}
```

Output:
10

```
class ABC; //Forward declaration of class
class XYZ
{
        int a;
        public:
        void setvalue(int x)
        {
                a=x;
        }
        friend void max(ABC,XYZ);
};
class ABC
{
        int b;
        public:
        void setvalue(int y)
        {
        b=y;
        }
        friend void max(ABC,XYZ); };
```

```
void max(ABC p, XYZ q)
{
        if(p.b>q.a)
        {
                cout<<"b is max";
        }
        else
        {
        cout<<"a is max"; }
}
void main()
{
ABC p;
XYZ q;
clrscr();
p.setvalue(20);
q.setvalue(10);
max(p,q);
getch();
}
```

Output:

```
20
```

Friend Class

- A friend class can access both private and protected members of the class in which it has been declared as friend.
- We can also use a friend Class in C++ using the friend keyword.
- **Syntax: friend class ClassName;**

```
class A
{
        int x =5;
        friend class B;
};
class B
{
        public:
        void display(A &a)
        {
        cout<<" x is : "<<a.x;
        }
};
void main()
{
        A a;
        B b;
        b.display(a);
        return 0;
}
```

```
Output:
X is : 5
```